# Connecting a Logical Framework to a First-Order Logic Prover (Extended Version)[⋆]

Andreas Abel, Thierry Coquand, and Ulf Norell

Department of Computing Science, Chalmers University of Technology
{abel,coquand,ulfn}@cs.chalmers.se

**Abstract.** We present one way of combining a logical framework and first-order logic. The logical framework is used as an interface to a first-order theorem prover. Its main purpose is to keep track of the structure of the proof and to deal with the high level steps, for instance, induction. The steps that involve purely propositional or simple first-order reasoning are left to a first-order resolution prover (the system Gandalf in our prototype). The correctness of this interaction is based on a general meta-theoretic result. One feature is the simplicity of our translation between the logical framework and first-order logic, which uses implicit typing. Implementation and case studies are described.

## Introduction

We work towards human-readable and machine-verifiable *proof documents* for mathematics and computer science. As argued by de Bruijn [dB80], dependent type theory offers an ideal formal system for representing reasoning steps, such as introducing parameters or hypotheses, naming constants or lemmas, using a lemma or a hypothesis. Type theory provides explicit notations for these proof steps, with good logical properties. Using tools like Coq [BC04], Epigram [AMM05], or Agda [CC99] these steps can be performed interactively. But low level reasoning steps, such as simple propositional reasoning, or equality reasoning, substituting equals for equals, are tedious if performed in a purely interactive way. Furthermore, propositional provers, and even first-order logic (FOL) provers are now very efficient. It is thus natural to create interfaces between logical frameworks and automatic propositional or first-order provers [BHdN02,ST95,MP04]. But, in order to arrive at proof documents which are still readable, only *trivial* proof steps should be handled by the automatic prover. Since different readers might have different notions of *trivial*, the automatic prover should not be a black box. With some effort by the human, the output of the prover should be understandable.

In this paper, we are exploring connections between a logical framework $\mathsf{MLF}_{\mathsf{Prop}}$ based on type theory and resolution-based theorem provers. One prob-

⋆ Research supported by the coordination action *TYPES* (510996) and thematic network *Applied Semantics II* (IST-2001-38957) of the European Union and the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

lem in such an interaction is that resolution proofs are hard to read and understand in general. Indeed, resolution proof systems work with formulæ in clause normal form, where clauses are (the universal closures of) disjunctions of literals, a literal being an atom or a negated atom. The system translates the negation of the statement to be proved to clause form, using skolemisation and disjunctive normal form. It then generates new clauses using resolution and paramodulation, trying to derive a contradiction. If successful, the system does pruning on the (typically high number of) generated clauses and outputs only the relevant ones.[1]

We lose the structure of the initial problem when doing skolemisation and clausification. Typically, a problem such as

$$\forall x.\exists y.\forall z.R(x,y) \Rightarrow R(x,z) \tag{1}$$

is negated and translated into the two contradictory unit clauses

$$\forall y.\ R(a,y), \qquad \forall y.\neg R(a,f(y)), \tag{2}$$

but the connection between the statement (1) and the refutation of (2) is not so intuitive.

We do not solve this problem here, but we point out that, if we restrict ourselves to implicitely universally quantified propositional formulæ, in the following called *open* formulæ, this problem does not arise. Furthermore, when we restrict to this fragment, we can use the idea of implicit typing [Bee05,WM89]. In this way, the translation from framework types to FOL formulæ is particularly simple. Technically, this is reflected by a general meta-theorem which ensures that we can lift a first-order resolution proof to a framework derivation. If we restrict the class of formulæ further to so-called *geometrical* open formulæ [CLR01,BC03], then the translation to clausal form is transparent. Indeed, any resolution proof for this fragment is intuitionistically valid and can be interpreted as it is in type theory. This meta-theorem is also the theoretical justification for our interface between $\mathsf{MLF}_{\mathsf{Prop}}$ and a resolution-based proof system.

We have implemented a prototype version of a type system in Haskell, with a connection to the resolution prover Gandalf [Tam97]. By restricting ourselves to open formulæ we sacrifice proof strength, but preliminary experiments show that the restriction is less severe than it may seem at first since the steps involving quantification are well handled at the framework level. Also, the proof traces produced by Gandalf are often readable (and surprisingly clever in some cases).

We think that we can represent Leslie Lamport proof style [Lam93] rather faithfully in this system. The high level steps such as introduction of hypotheses, case analysis, induction steps are handled at the framework level, and only the trivial steps are sent to the FOL automatic prover.

One can think also of other plug-in extensions, e.g., rewriting systems and computer algebra systems. We have experimented with a QuickCheck [CH00]

[1] If the search is not successful, it is quite hard to get any relevant information from the clauses that are generated. We have not yet analyzed the problem of getting useful feedback in this case.

plug-in, that allows random testing of some propositions. In general, each plug-in extension of our logical framework should be justified in the same way as the one we present in this paper: we prove a conservativity result which ensures that the use of this plug-in can be, if desired, replaced by a direct proof in the framework. This way of combining various systems works in practice, as suggested by preliminary experiments, and it is theoretically well-founded.

This paper is organized as follows. We first describe the logical framework $\mathsf{MLF}_{\mathsf{Prop}}$. We then present the translation from some LF types to FOL formulæ. The main technical result is then a theorem that shows that any resolution and paramodulation step, with one restriction, can be lifted to the framework level. Finally, we present some examples and extensions, and a discussion of related work.

## 1 The Logical Framework $\mathsf{MLF}_{\mathsf{Prop}}$

This section presents an extension of Martin-Löf's logical framework [NPS00] by propositions and local definitions.

*Expressions (terms and types).* We assume countable sets of variables $\mathsf{Var}$ and constants $\mathsf{Const}$. Furthermore, we have a finite number of built-in constants to construct the primitives of our type language. A priori, we do not distinguish between terms and types. The syntactic entities of $\mathsf{MLF}_{\mathsf{Prop}}$ are given by the following grammar.

| | | | |
|---|---|---|---|
| $\mathsf{Var}$ | $\ni x, y, z$ | | variables |
| $\mathsf{Const}$ | $\ni c, f, p$ | | constants |
| $\mathsf{BuiltIn}$ | $\ni \hat{c}$ | $::= \mathsf{Fun} \mid \mathsf{El} \mid \mathsf{Set} \mid () \mid \mathsf{Prf} \mid \mathsf{Prop}$ | built-in constants |
| $\mathsf{Exp}$ | $\ni r, s, P, Q$ | $::= \hat{c} \mid c \mid x \mid \lambda x r \mid r\, s \mid \mathsf{let}\, x\!:\!T\!=\!r\, \mathsf{in}\, s$ | expressions |
| $\mathsf{Ty}$ | $\ni T, U$ | $::= \mathsf{Set} \mid \mathsf{El}\, s \mid \mathsf{Prop} \mid \mathsf{Prf}\, P \mid \mathsf{Fun}\, T\, (\lambda x U)$ | types |
| $\mathsf{Cxt}$ | $\ni \Gamma$ | $::= \diamond \mid \Gamma, x\!:\!T$ | typing contexts |
| $\mathsf{Sig}$ | $\ni \Sigma$ | $::= \diamond \mid \Sigma, c\!:\!T \mid \Sigma, c\!:\!T\!=\!r$ | signatures |

We identify terms and types up to $\alpha$-conversion and adopt the convention that in contexts $\Gamma$, all variables must be distinct; hence, the context extension $\Gamma, x\!:\!T$ presupposes $(x\!:\!U) \notin \Gamma$ for any $U$. Similarly, a constant $c$ may not be declared in a signature twice. We abbreviate a sequence of context entries $x_1\!:\!T, \ldots, x_n\!:\!T$ of the same type by $x_1, \ldots, x_m\!:\!T$. Multiple application $r\, s_1 \ldots s_n$ is expressed as $r\, \boldsymbol{s}$. (Capture-avoiding) substitution of $r$ for $x$ in $s$ is written as $s[r/x]$, or $s[r]$ if $x$ is clear from the context of discourse.

For dependent function types $\mathsf{Fun}\, T\, (\lambda x U)$ we introduce the notation $(x : T) \to U$. Curried functions spaces $(x_1\!:\!T_1) \to \ldots (x_k\!:\!T_k) \to U$ are shortened to $(x_1\!:\!T_1, \ldots, x_k\!:\!T_k) \to U$, which explains the notation $(\Gamma) \to U$. Non-dependent functions $(\_\!:\!T) \to U$ are written $T \to U$. The inhabitants of $\mathsf{Set}$ are type codes; $\mathsf{El}$ maps type codes to types. E. g., $(a : \mathsf{Set}) \to \mathsf{El}\, a \to \mathsf{El}\, a$ is the type of the polymorphic identity $\lambda a \lambda x x$. Similarly $\mathsf{Prop}$ contains formal propositions $P$ and $\mathsf{Prf}\, P$ proofs of $P$.

Types of the shape $(\Gamma) \to \mathsf{Prf}\, P$ are called *proof types.* A context $\Gamma := x_1 : T_1, \ldots, x_n : T_n$ is a *set context* if and only if all $T_i$ are of the form $(\Delta) \to \mathsf{El}\ S$. In particular, if $P : \mathsf{Prop}$, then the proof type $(\Gamma) \to \mathsf{Prf}\, P$ corresponds to a universal first-order formula $\forall x_1 \ldots \forall x_n P$ with quantifier-free kernel $P$.

*Judgements.* The type theory $\mathsf{MLF}_{\mathsf{Prop}}$ is presented via five judgements, which are all relative to a (user-defined) signature $\Sigma$.

| | |
|---|---|
| $\Gamma \vdash_\Sigma$ | $\Gamma$ is a well-formed context |
| $\Gamma \vdash_\Sigma T$ | $T$ is a well-formed type |
| $\Gamma \vdash_\Sigma r : T$ | $r$ has type $T$ |
| $\Gamma \vdash_\Sigma T = T'$ | $T$ and $T'$ are equal types |
| $\Gamma \vdash_\Sigma r = r' : T$ | $r$ and $r'$ are equal terms of type $T$ |

All five judgements are defined simultaneously. Since the signature remains fixed in all judgements we will omit it. In this article, we only spell out the typing rules (see appendix). Judgmental type and term equality are generated from expansion of signature definitions as well as from $\beta$-, $\eta$-, and let-equality, the latter of which is given by $(\mathsf{let}\, x : T = r\, \mathsf{in}\, s) = s[r/x]$. The rules for equality are similar to the ones of $\mathsf{MLF}_\Sigma$ [AC05], and type-checking of normal terms with local definitions is decidable.

*Natural deduction.* We assume a signature $\Sigma_{\mathsf{nd}}$ (see appendix) which assumes the infix logical connectives $op ::= \wedge, \vee, \Rightarrow$, plus the defined ones, $\neg$ and $\Leftrightarrow$. Furthermore, it contains a set $\mathsf{PredSym}$ of basic predicate symbols $p$ of type $(\Gamma) \to \mathsf{Prop}$ where $\Gamma$ is a (possibly empty) set context. Currently we only assume truth $\top$, absurdity $\bot$, and typed equality $\mathsf{Id}$, but user defined signatures can extend $\mathsf{PredSym}$ by their own symbols. For each logical constructs, there are appropriate proof rules, e. g., a constant $\mathsf{impl} : \ (P, Q : \mathsf{Prop}) \to (\mathsf{Prf}\, P \to \mathsf{Prf}\, Q) \to \mathsf{Prf}\ (P \Rightarrow Q)$.

First-order logic assumes that every set is non-empty, and our use of a first-order prover is only sound under this assumption. Hence, we add a special constant $\epsilon : (D : \mathsf{Set}) \to \mathsf{El}\ D$ to $\Sigma_{\mathsf{nd}}$ which enforces this fact. Notice that this implies that all set contexts are inhabited[2].

Classical reasoning can be performed in the signature $\Sigma_{\mathsf{class}}$, which we define as the extension of $\Sigma_{\mathsf{nd}}$ by $\mathsf{EM} : (P : \mathsf{Prop}) \to \mathsf{Prf}\ (P \vee \neg P)$, the law of the excluded middle.

*The* FOL *rule.* This article investigates conditions under which the addition of the following rule is conservative over $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{nd}}$ and $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$, respectively.

$$\textsc{fol}\ \frac{\Gamma \vdash T}{\Gamma \vdash () : T}\ \Gamma \vdash_{\mathsf{FOL}} T$$

[2] Semantically, it may be fruitful to think of terms of type Set as inhabited Partial Equivalence Relations, while terms of type Prop are PERs with at most one inhabitant.

The side condition $\Gamma \vdash_{\mathsf{FOL}} T$ expresses that $T$ is a proof type and that the first-order prover can deduce the truth of the corresponding first-order formula from the assumptions in $\Gamma$. It ensures that only tautologies have proofs in $\mathsf{MLF}_{\mathsf{Prop}}$, but it is not considered part of the type checking. Meta-theoretical properties of $\mathsf{MLF}_{\mathsf{Prop}}$ like decidability of equality and type-checking hold independently of this side condition.

Conservativity fails if we have to compare proof objects during type-checking. This is because the rule FOL produces a single proof object for all (true) propositions, whereas upon removal of FOL the hole has to be filled with specific proof object. Hence two equal objects which each depend on a proof generated by FOL could become inequal after replacing FOL. To avoid this, it is sufficient to restrict function spaces $(x\!:\!T) \to U$: if $T$ is a proof type, then also $U$.

In the remainder of the paper, we use LF as a synonym for $\mathsf{MLF}_{\mathsf{Prop}}$.

## 2 Translation from $\mathsf{MLF}_{\mathsf{Prop}}$ to FOL

We shall define a *partial* translation from some LF types to FOL propositions. We translate only types of the form

$$(x_1\!:\!T_1, \ldots, x_k\!:\!T_k) \to \mathsf{Prf}\ (P(x_1, \ldots, x_k)),$$

and these are translated to *open* formulæ $[P(x_1, \ldots, x_k)]$ of first-order logic. All the variables $x_1, \ldots, x_k$ are considered universally quantified. For instance,

$$(x\!:\!\mathsf{El}\ \mathsf{N}) \to \mathsf{Prf}\ (\mathsf{Id}\ \mathsf{N}\ x\ x \wedge \mathsf{Id}\ \mathsf{N}\ x\ (\mathsf{add}\ 0\ x))$$

will be translated to $x = x \ \wedge\ x = \mathsf{add}\ 0\ x$. If we have a theory of lattices, that is, we have added

$$\begin{array}{ll} D & : \mathsf{Set} \\ \mathsf{sup} & : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{El}\ D \\ \leq & : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop} \end{array}$$

to the current signature, then $(x, y\!:\!\mathsf{El}\ D) \to \mathsf{Prf}\ (\mathsf{sup}\ x\ y \leq x \Leftrightarrow y \leq x)$ would be translated to $\mathsf{sup}\ x\ y \leq y \ \Leftrightarrow\ y \leq x$.

The translation is done at a syntactical level, without using types. We will demonstrate that we can lift a resolution proof of a translated formula to a LF derivation in the signature $\Sigma_{\mathsf{class}}$ (or in $\Sigma_{\mathsf{nd}}$, in some cases).

### 2.1 Formal Description of the Translation

We translate *normal* expressions, which means that all definitions have been unfolded and all redexes reduced. Three classes of normal $\mathsf{MLF}_{\mathsf{Prop}}$-expressions are introduced: (formal) *first-order terms* and (formal) *first-order formulæ*, which are quantifier free formulæ over atoms possibly containing free term variables, and *translatable formulæ*, which are first-order formulæ prefixed by quantification over set elements.

|  |  |  |
|---|---|---|
| $t, u$ | $::= x \mid f\,\boldsymbol{t}$ | first-order terms |
| $A, B$ | $::= p\,\boldsymbol{t} \mid \mathsf{Id}\,S\,t_1\,t_2$ | atoms |
| $W$ | $::= A \mid W\ op\ W'$ | first-order formulæ |
| $\phi$ | $::= (\varDelta) \to \mathsf{Prf}\,W$ | translatable formulæ ($\varDelta$ set context) |

*Proper terms* are those which are not just variables. For the conservativity result the following fact about proper terms will be important: In a well-typed proper term, the types of its variables are uniquely determined. For this reason, a formal first-order term $t$ may neither contain a binder ($\lambda$ or $\mathsf{let}$) nor a variable which is applied to something, for instance, $x\,u$.

An example of a first-order formula is $W_{\mathsf{ex}} := \mathsf{Id}\,D\,x\,(f\,y) \Rightarrow (\mathsf{Less}\,x\,(f\,y) \Rightarrow \bot)$, which is well-typed in the extension $D : \mathsf{Set},\ f : \mathsf{El}\ D \to \mathsf{El}\ D,\ \mathsf{Less} : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop}$ of signature $\Sigma_{\mathsf{nd}}$.

On the FOL side, we consider a language with equality (=), one binary function symbol $\mathsf{app}$ and one constant for each constant introduced in the logical framework. Having an explicit "$\mathsf{app}$" allows partial application of function symbols.

Let $\varDelta = x_1 : T_1, \ldots, x_n : T_n$ be a set context. A type of the form

$$\phi := (\varDelta) \to \mathsf{Prf}\,W$$

is translated into a universal formula $[\phi] = \forall x_1 \ldots \forall x_n [W]$. The translation $[W]$ of first-order formulæ and the translation $\langle t \rangle$ of first-order terms depends on $\varDelta$ and is defined recursively as follows:

|  |  |  |
|---|---|---|
| $[W_1\ op\ W_2]$ | $:= [W_1]\ op\ [W_2]$ | logical connectives |
| $[\mathsf{Id}\,S\,t_1\,t_2]$ | $:= \langle t_1 \rangle = \langle t_2 \rangle$ | equality |
| $[p\,t_1 \ldots t_n]$ | $:= p(\langle t_1 \rangle, \ldots, \langle t_n \rangle)$ | predicates, including $\top, \bot$ |
| $\langle x_i \rangle$ | $:= x_i$ | variables in $\varDelta$ |
| $\langle x \rangle$ | $:= c_x$ | variables not in $\varDelta$ |
| $\langle c \rangle$ | $:= c$ | 0-ary functions |
| $\langle f\,t_1 \ldots t_n \rangle$ | $:= f(\langle t_1 \rangle, \ldots, \langle t_n \rangle)$ | n-ary functions |

where we write $f(t_1, \ldots, t_n)$ for $\mathsf{app}(\ldots \mathsf{app}(\mathsf{app}(f, t_1), t_2), \ldots, t_n)$. Note that the translation is purely syntactical, and does not use type information. It is even homomorphic with two exceptions: (a) the typed equality of $\mathsf{MLF}_{\mathsf{Prop}}$ is translated into the untyped equality of FOL, and (b) variables bound outside $\phi$ have to be translated as constants.

For instance, the formula $(y : \mathsf{El}\ D) \to W_{\mathsf{ex}}$ is translated as $\forall y.\ c_x = f(y) \Rightarrow (\mathsf{Less}(c_x, f(y)) \Rightarrow \bot)$. Examples of types that cannot be translated are

$$(x : \mathsf{Prop}) \to \mathsf{Prf}\ x, \quad \mathsf{Prf}\ (F\ (\lambda x x)), \quad (y : \mathsf{El}\ D \to \mathsf{El}\ D) \to \mathsf{Prf}\ (P\ (y\ x)).$$

We shall also use the class of *geometrical formulæ*, given by the following grammar:

|  |  |
|---|---|
| $G ::= H \mid H \to G \mid G \wedge G$ | geometrical formula |
| $H ::= A \mid H \wedge H \mid H \vee H$ | positive formula |

The above example $W_{\mathsf{ex}}$ is geometrical. As we will show, (classical) first-order proofs of geometrical formulæ can be mapped to intuitionistic proofs in the logical framework with $\Sigma_{\mathsf{nd}}$.

## 2.2 Resolution Calculus

It will be convenient to use the following non-standard presentation of the resolution calculus [Rob65]. A *clause* $C$ is an open first-order formula of the form

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B_1 \vee \cdots \vee B_m$$

where we can have $n = 0$ or $m = 0$ and $A_i$ and $B_j$ are atomic formulæ. Following Gentzen [Gen35], we write such a clause on the form

$$A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m,$$

that is, $X \Rightarrow Y$, where $X$ and $Y$ are finite sets of atomic formulæ. An empty $X$ is interpreted as truth, an empty $Y$ as absurdity.

Resolution is forward reasoning. Figure 1 lists the rules for extending the current set of derived clauses: if all clauses mentioned in the premise of a rule are present, this rule can fire and the clause of the conclusion is added to the clause set.

$$\textsc{ax}\ \frac{}{A \Rightarrow A} \qquad \textsc{sub}\ \frac{X' \supseteq X \qquad X \Rightarrow Y \qquad Y \subseteq Y'}{X' \Rightarrow Y'}$$

$$\textsc{res}\ \frac{X_1 \Rightarrow Z_1, Y_1 \qquad X_2, Z_2 \Rightarrow Y_2}{(X_1, X_2 \Rightarrow Y_1, Y_2)\sigma}\ \sigma = \mathsf{mgu}(Z_1, Z_2)$$

$$\textsc{refl}\ \frac{}{\cdot \Rightarrow x = x} \qquad \textsc{para}\ \frac{X_1 \Rightarrow t = u, Y_1 \qquad X_2[t'] \Rightarrow Y_2[t']}{(X_1, X_2[u] \Rightarrow Y_1, Y_2[u])\sigma}\ \sigma = \mathsf{mgu}(t, t')$$

**Fig. 1.** Resolution calculus.

In our formulation, all rules are intuitionistically valid[3], and can be justified in $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{nd}}$. It can be shown, classically, that these rules are *complete* in the following sense: if a clause is a semantical consequence of other clauses then it is possible to derive it using the resolution calculus. Hence, any proof in FOL can be performed with resolution[4].

[3] In the standard formulation, the AX rule would read $\neg A \vee A$—the excluded middle.
[4] To deal with existential quantification we also need skolemisation.

It can be pointed out that the SUB rule is only necessary at the very end—any resolution proof can be normalized to a proof that only uses SUB in the final step.

Let the *restricted* paramodulation rule denote the version of PARA where both $t$ and $t'$ are proper terms (not variables).

## 2.3 Proof of Correctness

In this section, we show that every FOL proof of a translated formula $[\phi]$ can be lifted to a proof in $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$, provided the resolution proof confines to restricted paramodulation. This is not trivial because FOL is untyped and $\mathsf{MLF}_{\mathsf{Prop}}$ is typed, and our translation forgets the types. The crucial insight is that every resolution step preserves well-typedness.

Fix a signature $\Sigma$. A first-order term $t$ is *well-typed* iff there exists a context $\Delta$, giving types to the variables $x_1, \ldots, x_n$ of $t$, such that in the given signature, $\Delta \vdash t : T$ for some type $T$. For example, in the signature

$$\begin{array}{ll} D : \mathsf{Set} & f : \mathsf{El}\ D \to \mathsf{El}\ D \\ F : \mathsf{El}\ D \to \mathsf{Prop} & g : (x{:}\mathsf{El}\ D) \to \mathsf{Prf}\ (F\ x) \end{array}$$

the proper first-order terms $f\ x$, $F\ y$, and $g\ z$ are well-typed, but $F\ x\ y$ is not. Notice that if a *proper* FOL term is well-typed, then there is only one way to assign types to its variables.

**Lemma 1.** *If two proper first-order terms $t_1, t_2$ over disjoint variables are well-typed and unifiable, then the most general unifier $\mathsf{mgu}(t_1, t_2)$ is well-typed.*

For instance, $\mathsf{add}\ x\ 0$ and $\mathsf{add}\ (\mathsf{S}\ y)\ z$ are unifiable and well-typed and the most general unifier $\{x{\mapsto}\mathsf{S}\,y, z{\mapsto}0\}$ is well-typed. The lemma is proven in the appendix.

Using this lemma, we can lift any FOL resolution step to an LF resolution step. The same holds for any *restricted* paramodulation step, which justifies the translation of $\mathsf{Id}\ S\ t\ u$ as $\langle t\rangle = \langle u\rangle$ in FOL, Indeed, in the paramodulation step between $X_1 \Rightarrow t = u, Y_1$ and $X_2[t'] \Rightarrow Y_2[t']$ we unify $t$ and $t'$ and for Lemma 1 to be applicable both $t$ and $t'$ have to be proper terms. Similar arguments have been put forth by Beeson [Bee05] and Wick and McCune [WM89].

A clausal type is a formula which translates to a clause.

**Lemma 2.** *If two FOL clausal types $(\Gamma_1) \to \mathsf{Prf}\ (W_1)$ and $(\Gamma_2) \to \mathsf{Prf}\ (W_2)$ are derivable, and $C$ is a resolution of $[W_1]$ and $[W_2]$ then there exists a context $\Gamma$ and a derivable $(\Gamma) \to \mathsf{Prf}\ W$ such that $C = [W]$. The same holds if $C$ is derived from $[W_1]$ and $[W_2]$ by restricted paramodulation. Furthermore in both cases, $\Gamma$ is a set context if both $\Gamma_1$ and $\Gamma_2$ are set contexts.*

In the next theorems, $\phi, \phi_1, \ldots, \phi_k$ are translatable formulæ of the form $(\Gamma) \to \mathsf{Prf}\ W$ where $\Gamma$ is a set context.

The following theorem is a consequence of Lemma 2, since an open formula is (classically) equivalent to a conjunction of clauses.

**Theorem 3.** *If we can derive $[\phi]$ from $[\phi_1], \ldots, [\phi_k]$ by resolution and restricted paramodulation then $\phi$ is derivable from $\phi_1, \ldots, \phi_k$ in any extension of the signature $\Sigma_{\mathsf{class}}$.*

A resolution proof, as we have presented it, is intuitionistically valid. The only step which may not be intuitionistically valid is when we express the equivalence between an open formula and a conjunction of clauses. For instance the open formula $\neg P \vee Q$ is not intuitionistically equivalent to the clause $P \Rightarrow Q$ in general. This problem does not occur if we start with geometrical formulæ [BC03].

**Theorem 4.** *If we can derive $[\phi]$ from $[\phi_1], \ldots, [\phi_k]$ by resolution and restricted paramodulation and $\phi, \phi_1, \ldots, \phi_k$ are geometric formulæ, then $\phi$ is derivable from $\phi_1, \ldots, \phi_k$ in any extension of the signature $\Sigma_{\mathsf{nd}}$.*

It is important for the theorem that all set contexts are inhabited: if $D : \mathsf{Set}$ and $P : \mathsf{Prop}$ (with $x$ not free in $P$), then both

$$\phi_1 = (x : \mathsf{El}\ D) \to \mathsf{Prf}\ P \quad \text{and} \quad \phi_2 = \mathsf{Prf}\ P$$

are translated to the same FOL proposition $[\phi_1] = [\phi_2] = P$ but we can derive $\phi_2$ from $\phi_1$ in $\Sigma_{\mathsf{nd}}, D : \mathsf{Set}, P : \mathsf{Prop}$ only because $\mathsf{El}\ D$ is inhabited.

As noticed above, if we allow paramodulation from a variable, we could derive clauses that are not well-typed. For instance, in the signature

$$N_1 : \mathsf{Set}, 0 : \mathsf{El}\ N_1, h : (x : \mathsf{El}\ N_1) \to \mathsf{Prf}\ (\mathsf{Id}\ N_1\ x\ 0), A : \mathsf{Set}, a : \mathsf{El}\ A$$

the type of $h$ becomes $x = 0$ in FOL and from this we could derive, by paramodulation from the variable $x$, $a = 0$ which is not well-typed. This problem is also discussed in [Bee05,WM89] and the solution is simply to forbid the FOL prover to use paramodulation from a variable[5].

We can now state the conservativity theorem.

**Theorem 5.** *If a type is inhabited in the system $\mathsf{MLF}_{\mathsf{Prop}} + \textsc{fol} + \Sigma_{\mathsf{class}}$ then it is inhabited in $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$.*

*Proof.* By induction on the typing derivation, using Thm. 3 for FOL derivations.

## 2.4 Simple Examples

Figure 2 shows an extension of $\Sigma_{\mathsf{nd}}$ by natural numbers, induction and an addition function defined by recursion on the second argument. Now consider the goal $(x : \mathsf{El}\,\mathsf{N}) \to \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ 0\ x)\ x$. Using the induction schema and the propositional proof rules, we can give the proof term

$$\mathsf{indN}\ (\lambda x.\ \mathsf{Id}\,\mathsf{N}\,(\mathsf{add}\,0\,x)\,x)\ ()\ (\lambda a.\,\mathsf{impl}\,(\lambda ih\,()))$$

[5] This is possible in Otter. In Gandalf, this could be checked from the trace. Paramodulation from a variable is highly non-deterministic. For efficiency reasons, it was not present in some version of Gandalf, but it was added later for completeness. In the examples we have tried, this restriction is not a problem.

| | | |
|---|---|---|
| $\mathsf{N}$ | $: \mathsf{Set}$ | natural numbers |
| $0$ | $: \mathsf{El}\,\mathsf{N}$ | zero |
| $\mathsf{S}$ | $: \mathsf{El}\,\mathsf{N} \to \mathsf{El}\,\mathsf{N}$ | successor |
| $\mathsf{indN}$ | $: (P : \mathsf{El}\,\mathsf{N} \to \mathsf{Prop}) \to P\ 0$ $\to ((x : \mathsf{El}\,\mathsf{N}) \to P\ x \Rightarrow P\ (\mathsf{S}\ x))$ $\to (n : \mathsf{El}\,\mathsf{N}) \to P\ n$ | induction |
| $\mathsf{add}$ | $: \mathsf{El}\,\mathsf{N} \to \mathsf{El}\,\mathsf{N} \to \mathsf{El}\,\mathsf{N}$ | addition |
| $\mathsf{add0}$ | $: (x \ : \mathsf{El}\,\mathsf{N}) \to \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ x\ 0)\ x$ | axiom 1 of $\mathsf{add}$ |
| $\mathsf{addS}$ | $: (x, y : \mathsf{El}\,\mathsf{N}) \to \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ x\ (\mathsf{S}\ y))\ (\mathsf{S}\ (\mathsf{add}\ x\ y))$ | axiom 2 of $\mathsf{add}$ |

**Fig. 2.** A Signature of Natural Numbers and Addition.

in the logical framework, which contains these two FOL goals:

$$\begin{aligned} &\vdash_{\mathsf{FOL}} \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ 0\ 0)\ 0 \\ a : \mathsf{El}\,\mathsf{N},\ ih : \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ 0\ a)\ a &\vdash_{\mathsf{FOL}} \mathsf{Id}\ \mathsf{N}\ (\mathsf{add}\ 0\ (\mathsf{S}\ a))\ (\mathsf{S}\ a) \end{aligned}$$

Both goals can be handled by the FOL prover. The first goal becomes $\mathsf{add}\ 0\ 0 = 0$ and is proved from $\mathsf{add}\ x\ 0 = x$, the translation of axiom $\mathsf{add0}$. The second goal becomes $\mathsf{add}\ 0\ (\mathsf{S}\ a) = \mathsf{S}\ a$. This is a first-order consequence of the translated induction hypothesis $\mathsf{add}\ 0\ a = a$ and $\mathsf{add}\ x\ (\mathsf{S}\ y) = \mathsf{S}\ (\mathsf{add}\ x\ y)$, the translation of axiom $\mathsf{addS}$.

This example, though very simple, is a good illustration of the interaction between LF and FOL: the framework is used to handle the induction step and in the second goal, the introduction of the parameter $a$ and the induction hypothesis.

Here is another simple example which illustrates that we can call the FOL prover even in a context involving non first-order operations. This example comes from a correctness proof of Warshall's algorithm. Let $D : \mathsf{Set}$.

$$\begin{aligned} &F : \mathsf{El}\,D \to (\mathsf{El}\,D \to \mathsf{El}\,D \to \mathsf{Prop}) \to \mathsf{El}\,D \to \mathsf{El}\,D \to \mathsf{Prop} \\ &F\,a\,R\,x\,y = R\,x\,y \vee (R\,x\,a \wedge R\,a\,y) \\ &\mathit{swap} : (a, b, x, y : \mathsf{El}\,D) \to \mathsf{Prf}\ (F\,a\,(F\,b\,R)\,x\,y \Leftrightarrow F\,b\,(F\,a\,R)\,x\,y) \end{aligned}$$

The operation $F$ is a higher-order operation. However, in the context $R : \mathsf{El}\,D \to \mathsf{El}\,D \to \mathsf{Prop}$, the goal *swap* can be handled by the FOL prover. The normal form of $F\,a\,(F\,b\,R)\,x\,y \Leftrightarrow F\,b\,(F\,a\,R)\,x\,y$, where all defined constants (here only $F$) have been unfolded, is a translatable formula.

## 3 Implementation

To try out the ideas described in this paper we have implemented a prototype type checker in Haskell. In addition to the logical framework, the type checker supports implicit arguments and the extensions described in Section 6: sigma types, datatypes and definitions by pattern matching.

## 3.1 Implicit Arguments

A problem with LF as presented here is its rather heavy notation. For instance, to state that function composition is associative one would give the signature in Figure 3. This is very close to being completely illegible due to the fact that

$$
\begin{aligned}
&\mathit{comp} : (A, B, C : \mathsf{Set}) \to (\mathsf{El}\ B \to \mathsf{El}\ C) \to (\mathsf{El}\ A \to \mathsf{El}\ B) \to (\mathsf{El}\ A \to \mathsf{El}\ C)\\
&\mathit{comp}\ A\ B\ C\ f\ g = \lambda x.\ f\,(g\ x)\\[1em]
&\mathit{assoc} : \ (A, B, C, D : \mathsf{Set}) \to\\
&\qquad\quad (f : \mathsf{El}\ C \to \mathsf{El}\ D,\ g : \mathsf{El}\ B \to \mathsf{El}\ C,\ h : \mathsf{El}\ A \to \mathsf{El}\ B) \to\\
&\qquad\quad \mathsf{Prf}\ (\mathsf{Id}\ (\mathsf{El}\ A \to \mathsf{El}\ D)\ \ (\mathit{comp}\ A\ C\ D\ f\ (\mathit{comp}\ A\ B\ C\ g\ h))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{comp}\ A\ B\ D\ (\mathit{comp}\ B\ C\ D\ f\ g)\ h))
\end{aligned}
$$

**Fig. 3.** Associativity without Implicit Arguments.

we have to be explicit about the type arguments to the composition function. To solve the problem, we have implemented a mechanism for implicit arguments which allows the omission of arguments that can be inferred automatically. Using this mechanism the associativity example can be written as follows:

$$
\begin{aligned}
&(\circ)(A, B, C : \mathsf{Set}) : (\mathsf{El}\ B \to \mathsf{El}\ C) \to (\mathsf{El}\ A \to \mathsf{El}\ B) \to (\mathsf{El}\ A \to \mathsf{El}\ C)\\
&f \circ g = \lambda x.\ f\,(g\ x)\\[1em]
&\mathit{assoc}\ (A, B, C, D : \mathsf{Set}) :\\
&\qquad (f : \mathsf{El}\ C \to \mathsf{El}\ D,\ g : \mathsf{El}\ B \to \mathsf{El}\ C,\ h : \mathsf{El}\ A \to \mathsf{El}\ B) \to\\
&\qquad \mathsf{Prf}\ (f \circ (g \circ h) == (f \circ g) \circ h)
\end{aligned}
$$

In general, we write $x\,(\Delta) : T$ to say that $x$ has type $(\Delta) \to T$ with $(\Delta)$ implicit. The scope of the variables in $\Delta$ extends to the definition of $x$ (if there is one). For every use of $x$ we require that the instantiation of $(\Delta)$ can be inferred using pattern unification [Mil92]. Note that when we have implicit arguments we can replace $\mathsf{Id}$ with an infix operator $(==)$ $(D : \mathsf{Set}) : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop}$

We conjecture that the conservativity result can be extended to allow the omission of implicit arguments when translating to first-order logic if they can be inferred from the resulting first-order term. In this case we preserve the property that for a well-typed FOL term there exists a unique typing, which is an important lemma in the conservativity theorem. The kind of implicit arguments we work with can most often be inferred in this way. It is doubtful, however, that it would work for other kinds of implicit arguments such as implicit dictionaries used for overloading.

Omitting the implicit arguments, the formula $f \circ (g \circ h) = (f \circ g) \circ h$ in the context $A, B, C, D : \mathsf{Set}, f : \mathsf{El}\ C \to \mathsf{El}\ D, g : \mathsf{El}\ B \to \mathsf{El}\ C, h : \mathsf{El}\ A \to \mathsf{El}\ B$ is translated to

$$f \circ (g \circ h) = (f \circ g) \circ h$$

With this translation, the first-order proofs are human readable and, in many cases, correspond closely to a pen and paper proof.

## 3.2 The Plug-in Mechanism

The type checker is equipped with a general plug-in interface that makes it easy to experiment with connections to external tools. A plug-in should implement two functions: a *type checking function* which can be called on particular goals in the program, and a *finalization function* which is called after type checking.

To control where the type checking function of a plug-in is invoked we introduce a new form of expressions:

$$\mathsf{Exp} ::= \ldots \mid \mathit{name}{-}\mathbf{plugin}(s_1, \ldots, s_n) \qquad \text{invoking a plug-in}$$

where *name* is the name of a plug-in. It is possible to pass arguments $(s_1, \ldots, s_n)$ to the plug-in. These arguments can be arbitrary expressions which are ignored by the type checker. Hence it is possible to pass ill-typed terms as arguments to a plug-in; it is the responsibility of the plug-in to interpret the arguments. Most plug-ins, of course, expect well-typed arguments and in this case, the plug-in has to invoke the type checker explicitly on its arguments.

## 3.3 The FOL Plug-in

The connection between LF and FOL has been implemented as a plug-in using the mechanism described above. With this implementation we replace the built-in constant () by a call to the plug-in. The idea is that the plug-in should be responsible for checking the side condition $\Gamma \vdash_{\mathsf{FOL}} P$ in the FOL rule.

An important observation is that decidability of type checking and equality do not depend on the validity of the propositions being checked by the FOL plug-in—nothing will break if the type checker is led to believe that there is an $s : \mathsf{Prf}\bot$. This allows us to delay all first-order reasoning until after type checking. The rationale for doing this is that type checking is cheap and first-order proving is expensive.

Another observation is that it is not feasible to pass the entire context to the prover. Typically, the context contains lots of things that are not needed for the proof, but would rather overwhelm the prover. To solve this problem, we require that any axioms or lemmas needed to prove a particular goal are passed as arguments to the plug-in. This might seem a severe requirement, but bear in mind that the plug-in is intended for simple goals where you already have an idea of the proof.

More formally, the typing rule for calls to the FOL plug-in is

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash s_1 : \phi_1 \ \ldots\ \Gamma \vdash s_n : \phi_n}{\Gamma \vdash \textbf{fol}\textbf{--}\textbf{plugin}(s_1, \ldots, s_n) : \phi}\ \phi_1, \ldots, \phi_n \vdash_{\mathsf{FOL}} \phi.$$

When faced with a call to a plug-in the type checker calls the type checking function of the plug-in. In this case, the type checking function of the FOL plug-in will verify that the goal is a translatable formula and that the arguments are well-typed proofs of translatable formulæ. If this is the case it will report success to the type checker and store away the side condition in its internal state. After type checking the finalization function of the FOL plug-in is called. For each constraint $\phi_1, \ldots, \phi_n \vdash_{\mathsf{FOL}} \phi$, this function verifies that $[\phi]$ is derivable from $[\phi_1], \ldots, [\phi_n]$ in the resolution calculus by translating the formulæ to clause normal form and feeding them to an external first-order prover (Gandalf, at the moment). If the prover does not manage to find a proof within the given time limit, the plug-in reports an error.

### 3.4 A QuickCheck Plug-in

The plug-in mechanism is sufficiently general to allow many different kinds of plug-ins. One such plug-in that we have added is a QuickCheck [CH00] plug-in that allows the LF user to generate and run random test cases for a certain class of propositions. The QuickCheck plug-in works in a similar way to the FOL plug-in, in that the main work is done during the finalization phase. A difference is that since QuickCheck is implemented in Haskell there is no need to call any external tools, we can simply include the QuickCheck implementation in the type checker.

For the QuickCheck plug-in there is no hope of being able to prove conservativity in the way we have done for the FOL plug-in—as is well known, testing can only prove the presence of bugs, never the absence. This raises the question of what the status of a tested proposition should be. Clearly, it should not have the same status as a proved proposition, since that would make the system unsound. Our solution is to define a signature $\Sigma_{\mathsf{qc}}$ containing a constant $\mathsf{Tested}$ for representing tested proposition together with proof rules for tested propositions:

$$\begin{array}{lll}
\mathsf{Tested} & : \mathsf{Prop} \to \mathsf{Prop} & \\
\mathsf{testI} & : (P : \mathsf{Prop}) \to \mathsf{Prf}\,P \to \mathsf{Prf}\,(\mathsf{Tested}\,P) & \\
\mathsf{testAndI} & : (P_1, P_2 : \mathsf{Prop}) \to & \mathsf{Prf}\,(\mathsf{Tested}\,P_1) \to \mathsf{Prf}\,(\mathsf{Tested}\,P_2) \to \\
 & & \mathsf{Prf}\,(\mathsf{Tested}\,(P_1 \wedge P_2)) \\
\vdots & &
\end{array}$$

This allows us to reason about propositions that have only been tested in a controlled way.

## 4 Examples

The code in this section has been type checked successfully by our prototype type checker. In fact, the typeset version is automatically generated from the actual code. The type checker can infer which types are Sets and which are Props, so we omit El and Prf in the types.

### 4.1 Relational Algebra

Natural numbers can be added to the framework by three new constants $Nat$, $zero$, $succ$ plus an axiom for mathematical induction.

$$\begin{array}{l}
Nat \in \mathcal{S}et \\
zero \in Nat \\
succ \in Nat \to Nat \\
indNat\,(P \in Nat \to \mathcal{P}rop) \in P\;zero \to ((n \in Nat) \to P\;n \to P\,(succ\;n)) \to \\
\qquad\qquad\qquad\qquad\qquad\quad (m \in Nat) \to P\;m
\end{array}$$

Now we fix a set $A$ and consider relations over $A$. We want to prove that the transitive closure of a symmetric relation is symmetric as well. We define the notion of symmetry and introduce a symbol for relation composition. We could define $R \circ R' = \lambda x \lambda z \exists z.\, x\ R\ y \wedge y\ R'\ z$, but here we only assume that a symmetric relation composed with itself is also symmetric.

$$\begin{array}{l}
A \in \mathcal{S}et \\
sym \in (A \to A \to \mathcal{P}rop) \to \mathcal{P}rop \\
sym\;R \equiv (x, y \in A) \to R\;x\;y \implies R\;y\;x \\
\\
(\,\circ\,) \in (A \to A \to \mathcal{P}rop) \to (A \to A \to \mathcal{P}rop) \to (A \to A \to \mathcal{P}rop) \\
axSymO \in (R \in A \to A \to \mathcal{P}rop) \to sym\;R \to sym\,(R\,\circ\,R)
\end{array}$$

We define a monotone chain of approximations $R^{(n)}$ (in the source: $R\,\hat{}\,n$) of the transitive closure, such that two elements will be related in the transitive closure if they are related in some approximation. The main lemma states that all approximations are symmetric, if $R$ is symmetric.

$$\begin{array}{l}
(\,\hat{}\,) \in (A \to A \to \mathcal{P}rop) \to Nat \to (A \to A \to \mathcal{P}rop) \\
axTc \in (R \in A \to A \to \mathcal{P}rop) \to (x, y \in A) \to (n \in Nat) \to \\
\qquad\qquad ((R\;\hat{}\;succ\;n)\;x\;y \Leftrightarrow (R\;\hat{}\;n)\;x\;y \vee ((R\;\hat{}\;n) \circ (R\;\hat{}\;n))\;x\;y) \\
\qquad \wedge\ ((R\;\hat{}\;zero)\;x\;y \Leftrightarrow R\;x\;y) \\
\\
main \in (R \in A \to A \to \mathcal{P}rop) \to sym\;R \to (n \in Nat) \to sym\,(R\;\hat{}\;n) \\
main\;R\;h \equiv indNat \\
\qquad\qquad\qquad \mathbf{fol{-}plugin}\,(h,\ axTc\;R)
\end{array}$$

$$(\lambda\, n\, ih \to \textbf{fol}\textbf{−}\textbf{plugin}\,(h,\ axSymO\,(R\ \hat{}\ n)\, ih,\ axTc\, R,\ ih))$$

Induction is performed at the framework level, base and step case are filled by Gandalf. Pretty printed, Gandalf produces the following proof of the step case:

| | | |
|---|---|---|
| (1) | $\forall xy.\ (R^{(n)} \circ R^{(n)})\, x\, y \Longrightarrow (R^{(n)} \circ R^{(n)})\, y\, x$ | |
| (2) | $\forall mxy.\ R^{(succ\, m)}\, x\, y \Longrightarrow (R^{(m)} \circ R^{(m)})\, x\, y \vee R^{(m)}\, x\, y$ | |
| (3) | $\forall mxy.\ (R^{(m)} \circ R^{(m)})\, x\, y \Longrightarrow R^{(succ\, m)}\, x\, y$ | |
| (4) | $\forall mxy.\ R^{(m)}\, x\, y \Longrightarrow R^{(succ\, m)}\, x\, y$ | |
| (5) | $\forall xy.\ R^{(n)}\, x\, y \Longrightarrow R^{(n)}\, y\, x$ | |
| (6) | $R^{(succ\, n)}\, a\, b$ | |
| (7) | $R^{(succ\, n)}\, b\, a \Longrightarrow \bot$ | |
| (8) | $(R^{(n)} \circ R^{(n)})\, a\, b \vee R^{(n)}\, a\, b$ | (2), (6) |
| (9) | $(R^{(n)} \circ R^{(n)})\, b\, a \vee R^{(n)}\, a\, b$ | (1), (8) |
| (10) | $R^{(n)}\, a\, b$ | (3), (7), (9) |
| (11) | $R^{(n)}\, b\, a$ | (5), (10) |
| (12) | $\bot$ | (4), (7), (11) |

The transitive closure is now defined as $TC\, R\, x\, y = \exists n.\, R^{(n)}xy$. To formalize this, we add existential quantification and its proof rules. The final theorem demostrates how existential quantification can be handled in the framework.

$$
\begin{aligned}
&\mathit{Exists}\,(A \in \mathcal{Set}) \in (A \to \mathcal{Prop}) \to \mathcal{Prop}\\
&\mathit{existsI}\,(A \in \mathcal{Set})(P \in A \to \mathcal{Prop}) \in (x \in A) \to P\, x \to \mathit{Exists}\, P\\
&\mathit{existsE}\,(A \in \mathcal{Set})(P \in A \to \mathcal{Prop})(C \in \mathcal{Prop}) \in\\
&\qquad \mathit{Exists}\, P \to ((x \in A) \to P\, x \to C) \to C
\end{aligned}
$$

$$
\begin{aligned}
&TC \in (A \to A \to \mathcal{Prop}) \to A \to A \to \mathcal{Prop}\\
&TC\ R\ x\ y \equiv \mathit{Exists}\,(\lambda\, n \to (R\ \hat{}\ n)\ x\ y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{thm} \in (R \in A \to A \to \mathcal{Prop}) \to \mathit{sym}\ R \to \mathit{sym}\ (TC\ R)\\
&\mathit{thm}\ R\ h\ x\ y \equiv \mathit{implI}\ (\lambda\, p \to\\
&\qquad\qquad \mathit{existsE}\ p\ (\lambda\, n\ q \to \mathit{existsI}\ n\, \textbf{fol}\textbf{−}\textbf{plugin}(q,\ \mathit{main}\ R\ h\ n)))
\end{aligned}
$$

### 4.2 Category Theory

One application of the FOL plug-in is to category theory. Typically, proofs in category contain a fair amount of symbolic manipulation, something which we can leave to the plug-in.

To reason about category theory we introduce the appropriate constants together with their axioms.

$$
\begin{aligned}
&\mathit{Obj} \in \mathcal{Set}\\
&\mathit{Hom} \in \mathit{Obj} \to \mathit{Obj} \to \mathcal{Set}
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{id}\,(a \in \mathit{Obj}) \in \mathit{Hom}\ a\ a \\
(\,\circ\,)\,(a,\ b,\ c \in \mathit{Obj}) \in \mathit{Hom}\ b\ c \to \mathit{Hom}\ a\ b \to \mathit{Hom}\ a\ c \\
\\
\mathit{axId1}\,(a,\ b \in \mathit{Obj}) \in (f \in \mathit{Hom}\ a\ b) \to f \ == \ \mathit{id} \circ f \\
\mathit{axId2}\,(a,\ b \in \mathit{Obj}) \in (f \in \mathit{Hom}\ a\ b) \to f \ == \ f \circ \mathit{id} \\
\\
\mathit{assoc}\,(a,\ b,\ c,\ d \in \mathit{Obj}) \in \\
\qquad (f \in \mathit{Hom}\ c\ d) \to (g \in \mathit{Hom}\ b\ c) \to (h \in \mathit{Hom}\ a\ b) \to \\
\qquad (f \circ g) \circ h \ == \ f \circ (g \circ h)
\end{array}
$$

Now we can define what it means for a morphism to be *epi* and prove that if the composition of two morphisms is epi then the first morphism must also be epi.

$$
\begin{array}{l}
\mathit{isEpi}\,(a,\ b \in \mathit{Obj}) \in \mathit{Hom}\ a\ b \to \mathcal{P}rop \\
\mathit{isEpi}\, f \equiv (c \in \mathit{Obj}) \to (g,\ h \in \mathit{Hom}\ b\ c) \to \\
\qquad\qquad g \circ f \ == \ h \circ f \implies g \ == \ h \\
\\
\mathit{epiI}\,(a,\ b \in \mathit{Obj})(f \in \mathit{Hom}\ a\ b) \in \mathit{isEpi}\, f \to \mathit{isEpi}\, f \\
\\
\mathit{prop}\,(a,\ b,\ c \in \mathit{Obj}) \in (f \in \mathit{Hom}\ b\ c) \to (k \in \mathit{Hom}\ a\ b) \to \\
\qquad\qquad\qquad\qquad \mathit{isEpi}\,(f \circ k) \implies \mathit{isEpi}\, f \\
\mathit{prop}\, f\ k \equiv \mathit{impI}\ (\lambda \mathit{epi\_kf} \to \mathbf{fol}{-}\mathbf{plugin}(\mathit{assoc},\ \mathit{epi\_kf}))
\end{array}
$$

Gandalf has no problem proving this (very simple) proposition and, more importantly, the proof that Gandalf produces is very close the proof we would write by hand. Pretty printed, the proof we get looks as follows.

$$
\begin{array}{lll}
(1) & \forall X\, Y\, Z.\ (X \circ Y) \circ Z = X \circ (Y \circ Z) & \\
(2) & \quad \forall X\, Y.\ X \circ (f \circ k) = Y \circ (f \circ k) \implies X = Y & \\
(3) & \qquad\quad g \circ f == h \circ f & \\
(4) & \qquad\quad g == h \implies \bot & \\
(5) & \quad\ \ \forall X.\ g \circ (f \circ X) == h \circ (f \circ X) & \{(1),(3)\} \\
(6) & \qquad\quad \bot & \{(2),(4),(5)\}
\end{array}
$$

See the appendix for an example involving algebra and induction.

## 5 Related Work

Smith and Tammet [ST95] also combine Martin-Löf type theory and first-order logic, which was the original motivation for creating the system Gandalf. The main difference to their work is that we use implicit typing and restrict to quantifier-free formulæ. An advantage is that we have a simple translation, and hence get a quite direct connection to resolution theorem provers. Hence, we can hope, and this has been tested positively in several examples, that the proof traces we get from the prover are readable as such and therefore can been used as a proof certificate or as feedback for the user. For instance, the user can formulate new lemmas suggested by this proof trace. We think that this aspect of readability is more important than creating an explicit proof term in type theory (which would actually be less readable). It should be stressed that our conservativity result contains, since it is constructive, an algorithm that can transform the resolution proof to a proof in type theory, if this is needed.

Huang et. al. [HKK$^+$94] present the design of $\Omega$-MKRP[6], a tool for the working mathematician based on higher-order classical logic, with a facility of proof planning, access to a mathematical database of theorems and proof tactics (called methods), and a connection to first-order automated provers. Their article is a well-written motivation for the integration of human and machine reasoning, where they envision a similar division of labor as we have implemented. We have, however, not addressed the problem of mathematical knowledge management and proof tactics.

Wick and McCune [WM89] list three options for connecting type systems and FOL: include type literals, put type functions around terms, or use implicit typing. We rediscovered the technique of implicit typing and found out later that it is present already in the work of Beeson [Bee05]. Our work shows that this can also be used with dependent types, which is not obvious a priori. Our formulation of the correctness properties, as a conservativity statement, requires some care (with the role of the sort $\mathsf{Prop}$), and is an original contribution.

Bezem, Hendriks, and de Nivelle [BHdN02] describe how to transform a resolution proof to a proof term for *any* first-order formula. However, the resulting proof terms are hard to read for a human because of the use of skolemisation and reduction to clausal forms. Furthermore, they restrict to a fixed first-order domain.

Hurd's work on a Gandalf-tactic for HOL [Hur99] is along the same lines. He translates untyped first-order HOL goals to clause form, sends them to Gandalf and constructs an LCF proof from the Gandalf output. In later work [Hur02,Hur03] he handles types by having two translations: the untyped translation, and a translation with explicit types. The typed translation is only used when the untyped translation results in an ill-typed proof.

JProver [SLKN01] is a connection-based intuitionistic theorem prover which produces proof objects. It has been integrated into NuPrl and Coq. The translation from type theory to first-order logic involves some heuristics when to include or discard type information. Unfortunately, the description [SLKN01] does not contain formal systems or correctness arguments, but focuses on the connection technology.

Jia Meng and Paulson [MP04] have carried out substantial experiments on how to integrate the resolution theorem prover Vampire into the interactive proof tool Isabelle. Their translation from higher-order logic (HOL) to first-order logic keeps type information, since HOL supports overloading via axiomatic type

[6] Markgraf Karl Refutation Procedure.

classes and discarding type information for overloaded symbols would lead to unsound reasoning. They claim to cut down the search space via type information, but this is also connected to overloading. The aim of their work is different to ours: while they use first-order provers to do as much automatic proofs and proof search as possible, we employ automation only to liberate the user from seemingly trivial proof steps.

In Coq, NuPrl, and Isabelle, the user constructs a proof via tactics. We provide type theory as a proof language in which the user writes down a proof skeleton, consisting of lemmas, scoped hypotheses, invokation of induction, and major proof steps. The first-order prover is invoked to solve (easy) subgoals. This way, we hope to obtain human-readable proof documents (see our examples).

## 6 Conclusion and Future Work

We have described the implementation of a logical framework with proof-irrelevant propositions and its connection to the first-order prover Gandalf. Soundness and conservativity of the connection have been established by general theorems.

It is natural to extend LF by sigma types, in order to represent, for instance, mathematical structures. The extension of the translation to FOL is straightforward, we simply add a new binary function symbols for representing pairs. A more substantial extension is the addition of data type and functions defined by case [NPS90]. In this extension, it is possible to represent each connective as a parameterized data type. Each introduction rule is represented by a constructor, and the elimination rules are represented by functions defined by cases. This gives a computational justification of each of the axioms of the signature $\Sigma_{nat}$. The extension of the translation to FOL is also straightforward: each defined equations for functions becomes a FOL equality. One needs also to express that each constructor is one-to-one and that terms with distinct constructors are distinct.

We plan to the extend the conservativity theorem to implicit arguments as presented in Section 3.1. We also think that we can extend our class of translatable formulæ, for instance, to include some cases of existential quantification.

One could think of adding more plug-ins, with the same principle that they are justified by a general meta-theorem. For instance, one could add a plug-in to a model checker, or a plug-in to a system with a decision procedure for Presburger arithmetic.

*Acknowledgments.* We thank the members of the Cover project, especially Koen Claessen for discussions on implicit typing and the clausification tool Santa for a uniform connection to FOL provers, and Grégoire Hamon for programming the clausifier of the FOL plug-in in a previous version.

## References

[AC05] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 23–38. Springer, April 2005.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[BC03] Marc Bezem and Thierry Coquand. Newman's lemma – a case study in proof automation and geometric logic. *Bulletin of the EATCS*, 79:86–100, 2003. Logic in Computer Science Column.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[Bee05] Michael Beeson. Otter-$\lambda$ home page, 2005. URL: http://mh215a.cs.sjsu.edu/.

[BHdN02] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3–4):253–275, 2002. Special Issue *Mechanizing and Automating Mathematics: In honour of N.G. de Bruijn.*

[CC99] Catarina Coquand and Thierry Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages (LFM'99)*, Paris, France, September 1999.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CLR01] Michel Coste, Henri Lombardi, and Marie-Françoise Roy. Dynamical methods in algebra: Effective Nullstellensätze. *Annals of Pure and Applied Logic*, 111(3):203–256, 2001.

[dB80] Niklas G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in combinatory logic, lambda calculus and formalism*, pages 579–606, London-New York, 1980. Academic Press. Reprinted in: Selected Papers on Automath, edited by R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer, Studies in Logic, vol. 133, pp. 141-161. North-Holland 1994.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[HKK$^+$94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg H. Siekmann. Omega-MKRP: A proof development environment. In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 788–792. Springer, 1994.

[Hur99] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99, Nice, France*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999.

[Hur02] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 2002, Proceedings*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138. Springer, 2002.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA'03)*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[Lam93] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., February 1993. Also appeared as SRC Research Report 94.

[Mil92] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 2004.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

[NPS00] Bengt Nordström, Kent Petersson, and Jan Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science*, volume 5. Oxford University Press, October 2000.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning - First International Joint Conference, IJCAR 2001, Siena, Italy, June 2001, Proceedings*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer, 2001.

[ST95] Jan M. Smith and Tanel Tammet. Optimized encodings of fragments of type theory in first-order logic. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 265–287. Springer, 1995.

[Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[WM89] Cynthia A. Wick and William McCune. Automated reasoning about elementary point-set topology. *Journal of Automated Reasoning*, 5(2):239–255, 1989.

## Appendix

## A Typing Rules of $\mathsf{MLF}_{\mathsf{Prop}}$

Figure 4 shows the typing rules of $\mathsf{MLF}_{\mathsf{Prop}}$. The rules FUN-F and FUN-I carry a side condition $(*)$ that ensures that no type can depend on a proof, which is needed for the conservativity theorem.

---

Wellformed contexts $\Gamma \vdash$.

$$\textsc{cxt-empty}\ \frac{}{\diamond \vdash} \qquad \textsc{cxt-ext}\ \frac{\Gamma \vdash T}{\Gamma, x\!:\!T \vdash}$$

Wellformed types $\Gamma \vdash T$.

$$\textsc{set-f}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Set}} \qquad \textsc{prop-f}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Prop}} \qquad \textsc{fun-f}\ \frac{\Gamma \vdash T \qquad \Gamma, x\!:\!T \vdash U}{\Gamma \vdash (x\!:\!T) \to U}\ (*)$$

$$\textsc{set-e}\ \frac{\Gamma \vdash r : \mathsf{Set}}{\Gamma \vdash \mathsf{El}\, r} \qquad \textsc{prop-e}\ \frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \mathsf{Prf}\, P}$$

Typing $\Gamma \vdash r : T$.

$$\textsc{cst}\ \frac{\Gamma \vdash \quad (c\!:\!T) \in \Sigma}{\Gamma \vdash c : T} \qquad \textsc{hyp}\ \frac{\Gamma \vdash \quad (x\!:\!T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{conv}\ \frac{\Gamma \vdash r : T \quad \Gamma \vdash T = U}{\Gamma \vdash r : U}$$

$$\textsc{fun-i}\ \frac{\Gamma, x\!:\!T \vdash r : U}{\Gamma \vdash \lambda x r : (x\!:\!T) \to U}\ (*) \qquad \textsc{fun-e}\ \frac{\Gamma \vdash r : (x\!:\!T) \to U \qquad \Gamma \vdash s : T}{\Gamma \vdash r\, s : U[s/x]}$$

$$\textsc{let}\ \frac{\Gamma \vdash r : T \qquad \Gamma \vdash s[r/x] : U}{\Gamma \vdash \mathsf{let}\, x\!:\!T\!=\!r\ \mathsf{in}\ s : U}$$

Side condition $(*)$: If $T$ is a proof type, then also $U$.

---

**Fig. 4.** $\mathsf{MLF}_{\mathsf{Prop}}$ rules for contexts and typing.

## B A Signature for Natural Deduction

Figure 5 shows the signature $\Sigma_{\mathsf{nd}}$ for natural deduction proofs. Negation $\neg$ and logical equivalence $\Leftrightarrow$ are examples of defined constants in a signature.

## C Welltypedness of Unifier

We say that the terms $t_1, \ldots, t_n$ *fit* a context $\Delta = (x_1\!:\!T_1, \ldots, x_n\!:\!T_n)$ in $\Gamma$ iff $\Gamma \vdash t_i : T_i[t_1, \ldots, t_{i-1}]$ for all $1 \leq i \leq n$.

**Lemma 1.** *If two proper first-order terms $t, u$ over disjoint variables are well-typed and unifiable, then the most general unifier* $\mathsf{mgu}(t, u)$ *is well-typed.*

---

Predicate symbols and logical connectives.

| | | |
|---|---|---|
| $\mathsf{Const} \supseteq \mathsf{PredSym} \ni p$ | $::= \top, \bot, \mathsf{Id}$ | predicate symbols |
| $\mathsf{Const} \supseteq \mathsf{LogOp} \ni op$ | $::= \wedge, \vee, \Rightarrow$ | binary logical connectives |

Formation rules for propositional logic.

| | | |
|---|---|---|
| $\top, \bot$ | $: \mathsf{Prop}$ | truth, absurdity |
| $\wedge, \vee, \Rightarrow$ | $: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$ | conj., disj., impl. |
| $\neg$ | $: \mathsf{Prop} \to \mathsf{Prop} \qquad = \lambda P.\, P \Rightarrow \bot$ | negation |
| $\Leftrightarrow$ | $: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} = \lambda P \lambda Q.\, (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ | logical equivalence |

Proof rules for propositional logic.

| | | |
|---|---|---|
| $\mathsf{trueI}$ | $: \mathsf{Prf}\, \top$ | |
| $\mathsf{falseE}$ | $: (P : \mathsf{Prop}) \to \mathsf{Prf}\, \bot \to \mathsf{Prf}\, P$ | |
| $\mathsf{andI}$ | $: (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\, P_1 \to \mathsf{Prf}\, P_2 \to \mathsf{Prf}\ (P_1 \wedge P_2)$ | |
| $\mathsf{andE}_i$ | $: (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\ (P_1 \wedge P_2) \to \mathsf{Prf}\, P_i$ | for $i \in \{1, 2\}$ |
| $\mathsf{orI}_i$ | $: (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\, P_i \to \mathsf{Prf}\ (P_1 \vee P_2)$ | for $i \in \{1, 2\}$ |
| $\mathsf{orE}$ | $: (P_1, P_2, Q : \mathsf{Prop}) \to \mathsf{Prf}\ (P_1 \vee P_2) \to$ $(\mathsf{Prf}\, P_1 \to \mathsf{Prf}\, Q) \to (\mathsf{Prf}\, P_2 \to \mathsf{Prf}\, Q) \to \mathsf{Prf}\, Q$ | |
| $\mathsf{impI}$ | $: (P, Q : \mathsf{Prop}) \to (\mathsf{Prf}\, P \to \mathsf{Prf}\, Q) \to \mathsf{Prf}\ (P \Rightarrow Q)$ | |
| $\mathsf{impE}$ | $: (P, Q : \mathsf{Prop}) \to \mathsf{Prf}\ (P \Rightarrow Q) \to \mathsf{Prf}\, P \to \mathsf{Prf}\, Q$ | |

Equality.

| | | |
|---|---|---|
| $\mathsf{Id}$ | $: (D : \mathsf{Set}) \to \mathsf{El}\, D \to \mathsf{El}\, D \to \mathsf{Prop}$ | typed equality |
| $\mathsf{refl}$ | $: (D : \mathsf{Set},\ x : \mathsf{El}\, D) \to \mathsf{Prf}\ (\mathsf{Id}\, D\, x\, x)$ | reflexivity |
| $\mathsf{subst}$ | $: (D : \mathsf{Set},\ P : \mathsf{El}\ D \to \mathsf{Prop},\ x, y : \mathsf{El}\ D) \to$ $\mathsf{Prf}\ (\mathsf{Id}\, D\, x\, y) \to \mathsf{Prf}\ (P\, x) \to \mathsf{Prf}\ (P\, y)$ | substitutivity |

---

**Fig. 5.** The signature $\Sigma_{\mathsf{nd}}$ for natural deduction.

The lemma is a consequence of the following stronger proposition: *If* $t_1, \ldots, t_n$ *and* $u_1, \ldots, u_n$ *are lists of terms that fit the same context $\Delta$ in $\Gamma$ and $\sigma$ is the most general substitution such that $t_i\sigma = u_i\sigma$ for $1 \le i \le n$, then $\Gamma \vdash \sigma(x) : A$ for all $(x : A) \in \Gamma$.*

Let $\Gamma \vdash t : A$ and $\Gamma' \vdash u : B$. Since $t$ and $u$ are proper terms and unifiable, $t = f(\boldsymbol{t})$ and $u = f(\boldsymbol{u})$ for some constant $f : (\Delta) \to C$. Hence, $\boldsymbol{t}$ and $\boldsymbol{u}$ fit $\Delta$ in $\Gamma, \Gamma'$, which is a valid context since $\Gamma$ and $\Gamma'$ are disjoint. Now the proposition implies that $\mathsf{mgu}(t, u)$ is well-typed.

*Proof (of the proposition).* We follow the steps of a simple unification algorithm and consider the unification problem

$$t_1 = u_1,\ \ldots,\ t_n = u_n$$

If both $t_1$ and $u_1$ are proper terms, they are of the form $f(a_1, \ldots, a_k)$ and $f(b_1, \ldots, b_k)$ and we get a simpler unification problem

$$a_1 = b_1,\ \ldots,\ a_k = b_k,\ t_2 = u_2,\ \ldots,\ t_n = u_n$$

If, for instance, $t_1$ is a variable $x$, and $x$ does not appear in $u_1$, we claim that all variables in $u_1$ have a type which is independent of $x$. This holds if $u_1$ is a variable, since the type of $u_1$ is the same as the one of $x$, but it also holds if $u_1$ is a proper term, since the type of the variables in $u_1$ are then determined by $u_1$ alone, and $x$ does not appear in $u_1$. We can hence assume that all these variables appear before $x$ in $\Gamma = \Gamma_1, x{:}T, \Gamma_2$. We then get the simpler unification problem in $\Gamma_1, \Gamma_2[u_1/x]$

$$t_2[u_1/x] = u_2[u_1/x],\ \ldots,\ t_n[u_1/x] = u_n[u_1/x]$$

We proceed in this way until we get an empty list in the context in which the most general unifier of the two terms is well-typed.

## D Example Involving Computer Algebra

An example from M. Beeson [Bee05]. This example illustrates how we can combine the interactive style of the logical framework, for instance for the induction steps, with the first-order logic plugin.

In this example we want to reason about existentially quantified propositions so we add some new constants to the signature.

$$\begin{aligned}
&\mathit{Exists}\ (A \in \mathcal{S}et) \in (A \to \mathcal{P}rop) \to \mathcal{P}rop\\
&\mathit{existsI}\ (A \in \mathcal{S}et) \in (P \in A \to \mathcal{P}rop) \to (x \in A) \to P\,x \to \mathit{Exists}\ P\\
&\mathit{existsE}\ (A \in \mathcal{S}et) \in (P \in A \to \mathcal{P}rop) \to \mathit{Exists}\ P \to\\
&\qquad\qquad (C \in \mathcal{P}rop) \to ((x \in A) \to P\,x \implies C) \to C
\end{aligned}$$

We also need natural numbers. For this use the datatype extensions which allows us to define recursive functions over the natural numbers. For instance, we can write a recursive proof of the induction principle.

**data** $\mathit{Nat} \in \mathcal{S}et$ **where**
  zero $\in \mathit{Nat}$
  succ $\in \mathit{Nat} \to \mathit{Nat}$

$$\begin{aligned}
&\mathit{indNat} \in (P \in \mathit{Nat} \to \mathcal{P}rop) \to P\,\mathsf{zero} \to\\
&\qquad ((n \in \mathit{Nat}) \to P\,n \implies P\,(\mathsf{succ}\,n)) \to
\end{aligned}$$

$$
\begin{aligned}
&\qquad\qquad (x \in \mathit{Nat}) \to P\,x\\
&\mathit{indNat}\ P\ a\ g\ \mathsf{zero} \equiv a\\
&\mathit{indNat}\ P\ a\ g\ (\mathsf{succ}\ n) \equiv \mathit{impE}\ (g\ n)\ (\mathit{indNat}\ P\ a\ g\ n)
\end{aligned}
$$

The goal of the example is to prove that in an integral ring, the only nilpotent element is zero. We start by defining what it means to be an integral ring.

$$
\begin{aligned}
&\mathit{isRing} \in (R \in \mathcal{Set}) \to (R \to R \to R) \to (R \to R \to R) \to\\
&\qquad\qquad\qquad\qquad (R \to R) \to R \to R \to \mathcal{Prop}\\
&\mathit{isRing}\ R\ (+)\ (*)\ \mathit{minus}\ \mathit{Zero}\ \mathit{One} \equiv\\
&\ (x \in R) \to (y \in R) \to (z \in R) \to\\
&\ \ (\,(x + y) == (y + x)\\
&\quad \wedge (x + \mathit{Zero}) == x\\
&\quad \wedge (x + (\mathit{minus}\,x)) == \mathit{Zero}\\
&\quad \wedge (x + (y + z)) == ((x + y) + z)\\
&\quad \wedge (x * (y + z)) == ((x * y) + (x * z))\\
&\quad \wedge ((y + z) * x) == ((y * x) + (z * x))\\
&\quad \wedge (x * \mathit{One}) == x\\
&\quad \wedge (\mathit{One} * x) == x\\
&\quad \wedge (x * (y * z)) == ((x * y) * z)\\
&\ \ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isIntegral} \in (R \in \mathcal{Set}) \to (R \to R \to R) \to R \to \mathcal{Prop}\\
&\mathit{isIntegral}\ R\ (*)\ \mathit{Zero} \equiv\\
&\qquad (x \in R) \to (y \in R) \to x * y == \mathit{Zero} \Longrightarrow\\
&\qquad\qquad\qquad\qquad x == \mathit{Zero} \vee y == \mathit{Zero}
\end{aligned}
$$

In the following we work on a particular (but abstract) integral ring.

$$
\begin{aligned}
&R \in \mathcal{Set}\\
&(+) \in R \to R \to R\\
&(*) \in R \to R \to R\\
&\mathit{minus} \in R \to R\\
&\mathit{Zero} \in R\\
&\mathit{One} \in R
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{axR} \in \mathit{isRing}\ R\ (+)\ (*)\ \mathit{minus}\ \mathit{Zero}\ \mathit{One}\\
&\mathit{axI} \in \mathit{isIntegral}\ R\ (*)\ \mathit{Zero}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{power} \in \mathit{Nat} \to R \to R\\
&\mathit{power}\ \mathsf{zero}\ x \equiv \mathit{One}\\
&\mathit{power}\ (\mathsf{succ}\ n)\ x \equiv (\mathit{power}\ n\ x) * x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isZero} \in R \to \mathcal{Prop}\\
&\mathit{isZero}\ x \equiv x == \mathit{Zero}
\end{aligned}
$$

```
$isNilpotent \in R \rightarrow \mathcal{P}rop$
$isNilpotent\; x \equiv Exists\; (\lambda\, n \rightarrow isZero\; (power\; n\; x))$
```

This is all we need to start the proof. First we prove some lemmas.

```
$lemCancel \in (x \in R) \rightarrow (y \in R) \rightarrow x\; +\; y\; ==\; y \implies isZero\; x$
$lemCancel\; x\; y \equiv$
  $impI\; (\lambda h \rightarrow$
            $\mathbf{let}\; rem \in isZero\; (x\; +\; (y\; +\; minus\; y))$
                $rem \equiv \mathbf{fol{-}plugin}(h, axR)$
            $\mathbf{in}$
              $\mathbf{fol{-}plugin}(rem, axR)$
        $)$
```

The proof of $Zero * x == Zero$ is not trivial (but can be done purely automatically if desired) so we give the main steps of one possible proof explicitely.

```
$lemZero \in (x \in R) \rightarrow isZero\; (Zero\; *\; x)$
$lemZero\; x \equiv$
  $\mathbf{let}\; rem1 \in Zero\; +\; One\; ==\; One$
      $rem1 \equiv \mathbf{fol{-}plugin}(axR)$
      $rem2 \in (Zero\; +\; One)\; *\; x\; ==\; Zero\; *\; x\; +\; One\; *\; x$
      $rem2 \equiv \mathbf{fol{-}plugin}(axR)$
      $rem3 \in Zero\; *\; x\; +\; One\; *\; x\; ==\; One\; *\; x$
      $rem3 \equiv \mathbf{fol{-}plugin}(axR, rem1, rem2)$
  $\mathbf{in}$
    $\mathbf{fol{-}plugin}(rem3, lemCancel)$

$lemOneZero \in (x \in R) \rightarrow One\; ==\; Zero \implies isZero\; x$
$lemOneZero\; x \equiv \mathbf{fol{-}plugin}(axR, lemZero)$
```

The main lemma is proved by induction explicitely at the framework level.

```
$prop \in R \rightarrow Nat \rightarrow \mathcal{P}rop$
$prop\; x\; n \equiv isZero\; (power\; n\; x) \implies isZero\; x$

$lemMain \in (x \in R) \rightarrow (n \in Nat) \rightarrow prop\; x\; n$
$lemMain\; x \equiv$
  $\mathbf{let}\; base \in prop\; x\; \mathsf{zero}$
      $base \equiv \mathbf{fol{-}plugin}(lemOneZero)$
      $step \in (n \in Nat) \rightarrow prop\; x\; n \implies prop\; x\; (\mathsf{succ}\; n)$
      $step\; n \equiv \mathbf{fol{-}plugin}(axR, axI)$
  $\mathbf{in}$
```

```
    $indNat\ (prop\ x)\ base\ step$

$thm \in (x \in R) \rightarrow isNilpotent\ x \rightarrow isZero\ x$
$thm\ x\ h \equiv existsE\ (\lambda n \rightarrow isZero\ (power\ n\ x))\ h\ (isZero\ x)\ (lemMain\ x)$
```